601.220 Intermediate Programming

Passing arrays to functions

Outline

- Review of `sizeof`
- Passing arrays to functions

Recall: `sizeof`

- How big is an `int` (on ugrad)?
- `sizeof` operator returns size in bytes

```
// sizeof_eg_1.c:
#include <stdio.h>
int main() {
    int int_bytes = sizeof(int);
    printf("# bytes in int = %d", int_bytes);
    return 0;
}
```

```
$ gcc sizeof_eg_1.c -std=c99 -pedantic -Wall -Wextra
$ ./a.out
# bytes in int = 4
```

Recall: `sizeof`

```c
// sizeof_eg_2.c:
#include <stdio.h>

int main() {
    printf("# bytes in char = %lu\n", sizeof(char));
    printf("# bytes in int = %lu\n", sizeof(int));
    printf("# bytes in float = %lu\n", sizeof(float));
    printf("# bytes in double = %lu\n", sizeof(double));
    return 0;
}
```

```
$ gcc sizeof_eg_2.c -std=c99 -pedantic -Wall -Wextra
$ ./a.out
# bytes in char = 1
# bytes in int = 4
# bytes in float = 4
# bytes in double = 8
```

sizeof with Arrays

How big is an array?

```
// sizeof_eg_3.c:
#include <stdio.h>

int main() {
    int days[30] = {0}; // initializes all elements to 0
    printf("# bytes in days array = %lu\n", sizeof(days));
    return 0;
}
```

```
$ gcc sizeof_eg_3.c -std=c99 -pedantic -Wall -Wextra
$ ./a.out
# bytes in days array = 120
```

4 bytes per `int`, 30 `int`s

`sizeof` with Strings

How big is a string?

```
// sizeof_eg_4.c:
#include <stdio.h>

int main() {
    char pet[] = "cat";
    printf("# bytes in pet string = %lu\n", sizeof(pet));
    return 0;
}
```

```
$ gcc sizeof_eg_4.c -std=c99 -pedantic -Wall -Wextra
$ ./a.out
# bytes in pet string = 4
```

1 byte per character, 4 characters (including terminator)

Function Arguments

- Recall we saw that functions pass arguments "by value" – a copy is made and assigned to the parameter variable local to the callee
- Changes made to local variables and parameters in callee are not visible to caller

Passing Arrays to Functions

- Extra care is required when passing arrays to functions, or returning them from functions
- Arrays are *not* passed by value
  - Copying could be excessive
- Instead, passing an array amounts to passing a *pointer to its first element*
  - A pointer is a variable which holds an address (we'll discuss these more next week)
- Callee *can modify* the array

Passing Arrays to Functions: Example 1

```c
// function_arrpass_eg1.c:
#include <stdio.h>
// No need to specify a length for array parameter itself. The same amount of info is passed
// whether array is size 6 or size 600 -- an 8-byte address.
// So we feed in 2nd parameter to tell function the array's length.
int total(int n[], int len) {

    int tot = 0;
    for(int i = 0; i < len; i++) {
        tot += n[i];
    }
    return tot;
}

int main() {
    int evens[6] = {0, 2, 4, 6, 8, 10};
    printf("%d\n", total(evens, 6));
    return 0;
}
```

```
$ gcc function_arrpass_eg1.c -std=c99 -pedantic -Wall -Wextra
$ ./a.out
30
```

Passing Arrays to Functions: Example 2

```c
// function_arrpass_eg2.c:
#include <stdio.h>

// Multiply each array element by factor, modifying the array
void scale_array(float arr[], int len, float factor) {
    for(int i = 0; i < len; i++) {
        arr[i] *= factor;
    }
}

int main() {
    float sequence[5] = {0.0, 1.0, 2.0, 3.0, 4.0};
    scale_array(sequence, 5, 2.0);
    for(int i = 0; i < 5; i++) {
        printf("sequence[%d] = %.1f\n", i, sequence[i]);
    }
    return 0;
}
```

Passing Arrays to Functions: Example 2 Output

```
$ gcc function_arrpass_eg2.c -std=c99 -pedantic -Wall -Wextra
$ ./a.out
sequence[0] = 0.0
sequence[1] = 2.0
sequence[2] = 4.0
sequence[3] = 6.0
sequence[4] = 8.0
```

Passing Arrays to Functions: Warning!!!

When you pass an array to a function, the function no longer knows its size.

Instead, it will return the size of the pointer. (More on this later.)

```
// function_array_sizeof.c:
#include <stdio.h>

// Print out the size of an array
void print_array_size( int arr[] ) {
    printf( "Array size (in function): %d\n" , sizeof(arr) );
}

int main() {
    int arr[1000];
    printf( "Array size (in main): %d\n" , sizeof(arr) );
    print_array_size( arr );
    return 0;
}
```

```
$ ./a.out
Array size (in main): 4000
Array size (in function): 8
```

Note: The compiler *will* warn you about this. Don't ignore the warning.

Checkpoint Question!

What is the output of the following program?

```c
#include <stdio.h>
void myFunc(int x, int a[]) {
  x += 3;
  a[0] = 42;
}
int main(void) {
  int y = 4;
  int r[] = { 1, 2, 3 };
  myFunc(y, r);
  printf("y=%d, r[0]=%d\n",
         y, r[0]);
  return 0;
}
```

A. `y=4, r[0]=1`

B. `y=7, r[0]=1`

C. `y=4, r[0]=42`

D. `y=7, r[0]=42`

E. None of the above

Returning an Array from a Function

- When returning an array, the return type is the array's base type with $*$ added
  - It's technically a *pointer*
- *However*, we don't yet know the correct way to return arrays

Returning an Array from a Function: Bad Example

```c
// function_arrpass_eg3.c:
#include <stdio.h>

double* scale_array(double arr[], double factor) {
    double scaled_arr[5];  //suppose we just know array's size is 5
    for(int i = 0; i < 5; i++) {
        scaled_arr[i] = arr[i] * factor;
    }
    return scaled_arr;
}
int main() {
    double array[] = {1.0, 4.5, 8.4, 2.5, 8.3};
    double* scaled_array = scale_array(array, 2.0);
    printf("%0.2f %0.2f\n", scaled_array[0], scaled_array[4]);
    return 0;
}
```

```
$ gcc function_arrpass_eg3.c -std=c99 -pedantic -Wall -Wextra
function_arrpass_eg3.c: In function ‘scale_array’:
function_arrpass_eg3.c:8:12: warning: function returns address o
    8 |     return scaled_arr;
      |            ^~~~~~~~~~
```

- error message says: *function returns address of local variable*

For Now, We Can Pass In An Empty Array To Fill

- Instead of returning a local array, caller should pass in "destination" array to modify, as we did here:

```
void scale_array(float arr[], int len, float factor) {
    for(int i = 0; i < len; i++) {
        arr[i] *= factor;
    }
}
```

Array Parameters That Shouldn't Be Modified

- When an array parameter *should not* be modified by the function, add `const` before the type
- Compiler gives an error if you try to modify a `const` variable

```c
// function_arrpass_eg4.c:
#include <stdio.h>

void scale_array(const float arr[], int len, float factor) {
//               ^^^^^
    for(int i = 0; i < len; i++) {
        arr[i] *= factor;
    }
}

int main() {
    float sequence[5] = {0.0, 1.0, 2.0, 3.0, 4.0};
    scale_array(sequence, 5, 2.0);
    return 0;
}
```

Array Parameters That Shouldn't Be Modified

```
$ gcc function_arrpass_eg4.c -std=c99 -pedantic -Wall -Wextra
function_arrpass_eg4.c: In function ‘scale_array’:
function_arrpass_eg4.c:6:16: error: assignment of read-only location ‘*
    6 |         arr[i] *= factor;
      |                ^~
```

- `arr` is "read-only" because of `const` in its type
  - Similar to `final` in Java
- We'll see an example of a `const` array parameter in today's exercise